Truly Adaptive Bloom Filters with Monotone Streaming Updates

Devan Shah
Princeton University

David Yan
Princeton University

Abstract

The bloom filter is an efficient and probabilistic insertion-only data structure for testing set inclusion that offers improved run time and lower memory at the cost of occasional false positives queries, i.e., falsely returning that an element is in the set. As the false positive rate directly correlates with the number of insertions, prior works [2, 6] leverage patterns in the inserted data to limit insertions in an underlying bloom filter, leading to dramatic performance improvements. However, typical applications of Bloom Filters, such as to content caching, malicious URLs, or checking redundant usernames, have high potential for distribution shifts, leaving models learned on initial data distributions irrelevant, or possibly detrimental, for future inserts or queries, dramatically degrading performance. Typical solutions for model adaption, such as continuous fine-tuning, training supplementary models, or retraining under model degradation, are not applicable to the bloom filter setting due to the unique guarantees required for the model updates. Moreover, in many settings, there are no true negative labels, rendering conventional training techniques impractical and prone to misleading models. In addition, we show existing model classes used to create learned bloom filters may be overeager, prone to over-classifying the key space leaving them especially vulnerable to distribution shifts and with a high attack surface, rendering the systems inenept for many use-cases.

To address these issues, in this work we introduce Tru-Adapt, a classification model for the Learned Bloom Filter [6] providing:

1. A novel adaptive bloom filter system that learns underlying distributions with ordinary filter use and handles distribution shift to reduce FPR without any initial distribution knowledge and with $O(\log(n))$ performance.

2. We illustrate that existing learned bloom filter approaches, despite leveraging data distribution insights, are over-eager in their modeling leaving them vulnerable to high FPR upon distribution shifts.

3. We provide algorithmic guarantees for performance in addition to rigorous evaluations on realistic workloads.

4. We provide a lightweight algorithm with minimal hyper-parameters and no domain expertise required, ensuring convenient maintenance and satisfactory performance.

5. We ensure that our provided solution can be used to extend existing classifiers into adaptive classifiers if there are existing accurate key-distribution models.

1 Introduction

1.1 Learned Bloom Filters

The bloom filter offers an efficient data structure for testing probabilistic set inclusion. For many workloads, the underlying set is only added elements, and developers would be willing to allow occasional false positives in return for increased performance. The bloom filter offers a data structure with this trade off and are commonly employed in conjunction with more complicated systems to avoid more expensive operations, such as cache reads for an item not in the cache.

The bloom filter on M bits is implemented with a bit array $m \in \{0,1\}^M$, initialized to $m = \vec{0}$, and k hash functions $h_1, h_2, \ldots, h_k$ which map to $\{0, \ldots, M-1\}$. For each added key e, we assign $m[h_i(e)] = 1 \ \forall i$. To query whether element e is in the filter, we test whether $m[h_i(e)] = 1 \ \forall i$. Clearly the bloom filter cannot have any false negatives, but as the same bits may be accessed and changed by multiple keys, it is possible for a non-key e all the required bits in m are set by other keys, leading to a false positive. As each hash function accesses a point independently at random, the probability of a false positive for any given key is $(\frac{\sum_i m_i}{M})^k$, although note that more hash functions increases the rate at which the underlying table m fills.

For a bloom filter on M bits, with optimal choice of k and a desired FPR rate of below 0.05, we may only insert $\frac{M}{5}$ items. However, for many bloom filter use cases, we may be able to take advantage of the key distribution to reduce inserts and thus improve filter performance. For instance, if keys included most integers between 50 and 100 and a few outliers, we could dramatically decrease filter utilization by only inserting the exceptions into the filter and immediately returning true for a query between 50 and 100. This maintains no false negatives, although model mis-classifications (i.e. for non-keys in $[50, 100]$) could potentially lead to increased FPR.

More generally, Kraska et. al [6], consider the setting where, based on prior filter queries, we have a training set of keys and non-keys we can train a discriminator model f on. Choosing a threshold $\tau \in [0, 1]$, we only insert a new key x into the bloom filter B if $f(x) < \tau$ and, when querying inclusion for x, we return true if either $f(x) > \tau$ or the filter B claims to contain x. As before, we can view f as a classifier for x being in or out of the set, and B as an overflow buffer to ensure the false negative rate is 0.

1.2 LBF Limitations

Kraska et. al's [6] proposed Learned Bloom Filter (LBF) offers considerable performance improvements over a traditional Bloom Filter, but assumes a powerful and static classifier modelbased on an adequate training set. For many applications of bloom filters, as we do not store the queries and keys for space efficiency, and so we do not have a large dataset of labelled key and non-keys. Moreover, once we provide a function f for the Learned Bloom Filter, we struggle to change the function f. This is as, for a new function f^*, to ensure we maintain a 0% False Negative Rate (FNR), there cannot be any keys x such that $f(x) > \tau$ yet $f^*(x) \leq \tau$. Otherwise this key, when considered by f would not be inserted into the underlying filter, but f^* would expect it to be. However, as we cannot store the set of keys x, we must ensure this invariant globally, i.e., for input domain $\mathcal{D}$ for $x \in \mathcal{D}$, $f(x) > \tau \Rightarrow f^*(x) > \tau$, which we reduce further to $\forall x \in \mathcal{D}, f^*(x) \geq f(x)$, a condition we refer to as "monotonicity" or "update monotonicity". Traditional model families and function classes are unable to provide such function updates.

Kraska et. al. and later authors [2] train random forest discriminator models and consider similar model classes. However, general space splitting models, and other common model classes, such as neural networks, tend to over-eagerly approximate the key distribution. As we explore later, training common models on low-dimensional realistic distributions can lead to these models classifying half the input space as keys, a vast over-estimate that would lead to terrible FPR performance under a query distribution shift into the extended domain. In the high dimensional case, these issues persist but

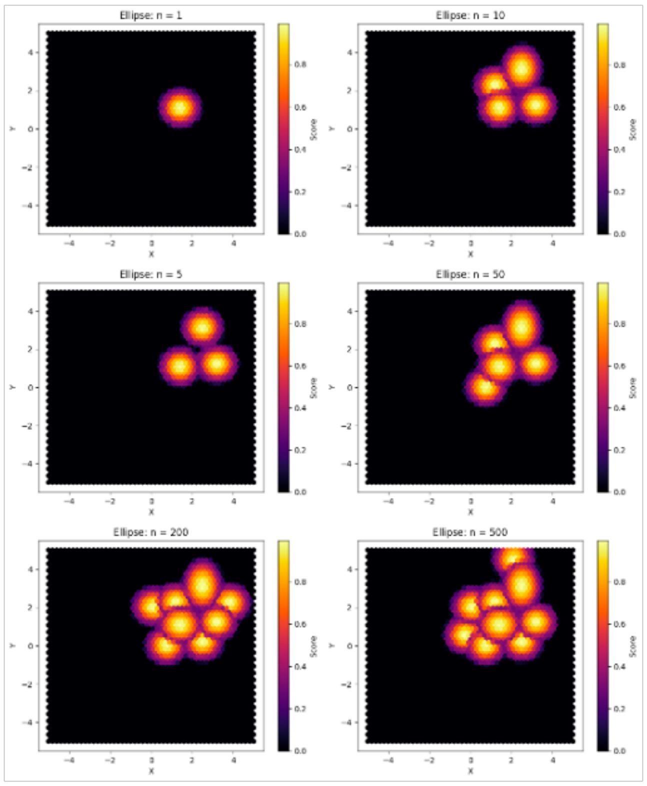

Figure 1: Heat-map of TruAdapt learning a pattern via Gaussian modeling over $n = 500$ inserts

would be challenging for a human evaluator to notice and remedy, potentially leading to sudden performance decay that cannot easily be addressed.

We introduce a custom model family and novel monotonic streaming learning algorithm that leverage patterns in the key embedding space to quickly adapt to distribution shifts and avoid the "over-eager" approximations common to existing models. In cases where existing models provide strong representations of the key distribution, our techniques can be applied in addition to the existing models to render these models adaptive.

In introducing streaming methods to handle distribution shift and avoid "over-eager" generalizations, we mitigate the pitfalls preventing learned bloom filters from achieving more widespread industry use and provide improved performance and easier implementation.

Our models f fundamentally approximate an energy function on the key-space via a set of Gaussian distributions, which thus provides an estimate of the key-space by choosing an appropriate threshold τ and considering $\{x \in \mathcal{D} \mid f(x) > \tau\}$. Our choice of model design ensures the desired monotonicity.

We provide three systems that are variations on the same core algorithm:

1. f models $\mathcal{K}$ via $\{x \in \mathcal{D} \mid f(x) > \tau\}$ with $O(\log(n))$ symmetric Gaussian distributions
2. f models $\mathcal{K}$ via $\{x \in \mathcal{D} \mid f(x) > \tau\}$ with $O(\log(n))$ general Gaussian distributions
3. f models $\mathcal{K}$ via $\{x \in \mathcal{D} \mid f(x) > \tau\}$ with $O(1)$ general Gaussian distributions

Where n refers to the amount of inserts. We note that typically bloom filters offer $O(1)$ evaluation, yet $O(\log(n))$ grows sufficiently small such that the performance is comparable to other learned bloom filters. Algorithm 1 and algorithm 3 are modifications of algorithm 2 that provide efficiency and memory improvements with empirically similar performance, and thus most of our analysis be on algorithm 2. These models are inspired by the success of Gaussian Splatting in modeling complex 3D density functions in computer vision [5] and by the success of Gaussian Mixture Models, which have shown great success in modeling complex distributions.

2 Problem Formalization

Let $f \in \mathcal{F}$ represent the learned model, $\mathcal{F}$ represent the model function class, S represent the true key set, B represent the bloom filter, $\mathcal{D}$ represent the input domain, which we assume to be bounded, $\mathcal{K} \subset \mathcal{D}$ be the key-space, and $\sigma : \mathcal{F} \times \mathcal{D} \to \mathcal{F}$ be the function update algorithm given an insertion. We aim to produce an algorithm with the high level structure equivalent to that in *Abstract Algorithm*, which is the standard Learned Bloom Filter outline with the addition of updates on line 7.

Algorithm 1 Abstract Algorithm

```
1: Insert:
2: if $f(x) > \tau$ then
3:     exit
4: end if
5: if $f(x) \leq \tau$ then
6:     $x \to B$
7:     $f := \sigma(f, x)$
8: end if
9: Query:
10: if $f(x) > \tau$ or $x \in B$ then
11:     return True
12: else
13:     return False
14: end if
```

Note that the Learned Bloom Filter discriminator learns a distribution over the input space that approximately corresponds to the probability an input is in the filter. With this viewpoint, and as we are considering inputs in the embedding space, our function f can be viewed as a Joint Energy-Based Model [4], with $E(x)$ defining the true energy function and thus, for some τ, $\mathcal{K} = \{x \in \mathcal{D} \mid E(x) > \tau\}$. We thus desire f that provides an accurate continuous approximation of $E(x)$. As we have limited knowledge of $E(x)$ and the true key space, we instead consider f that minimizes:

$$\mathcal{L}(f) = -\frac{1}{|S|}\sum_{x \in S} \mathbb{1}[f(x) > \tau] + \alpha\, \mathbb{E}_{x \sim \mathrm{Unif}(\mathcal{D})}[\mathbb{1}[f(x) > \tau]]$$

with α a weight impacting the contributions to the FPR from the function compared to from a crowded bloom filter. This loss function is also common in recommendation systems, which have a similar challenge of modeling sparse "keys" (user-object positive matchings) [3]. Note additionally that, by scaling f, we can assume $\tau = 1$. Moreover, as our evaluation of loss is entirely from $\mathbb{1}[f(x) > \tau]$, we can improve performance and extend our effective function class by instead directly modeling the regions $E(x) > \tau = 1$, learning instead the function $f_{ind} : \mathcal{D} \to \{0, 1\}$.

Additionally remark that if we have any prior knowledge of the distribution, such as a model $p(x)$, the function f and f_{ind} can be learned in order to adapt $p(x)$ monotonically by modeling instead $E(x) - p(x)$. Thus our approach can generalize to cases where there exist a strong prior on the distribution, such as if there already exists a strong model trained for an LBF system. We thus enable adaptive properties for pre-existing algorithms and allowing pre-existing models to adapt to a shifted energy functions, maximizing ease of integration and compatibility with existing learned systems.

3 Algorithm

3.1 Overview

For our algorithm, we make the assumption that x can be mapped by an embedding function to a semantically meaningful domain, which is true for many variable types, such as words, images, and neural-net classifiable inputs among others input types [4]. As bloom filter usage fundamentally derives from human usage, we assume recognizable patterns will occur at the semantic level. For the examples and discussion in [6], the assumptions made are similar in that the authors expect "*realistic queries*" (emph. in original) [6].

Thus, patterns can be viewed as a high dimensional segmentation, which may be conveniently representable as a series of clusters, with inspiration of the success of Gaussian Mixture Models in learning representations of complex distributions. Cluster-based models hold the advantage of being easily-adaptable and offering increased guarantees through greater interpretability.

To derive optimal model class for f_{ind}, we will first consider f. By the above, we will restrict to considering model families containing functions of the form, $f(x) = \max_{c \in \mathcal{C}} r(c, x)$, with $\mathcal{C}$ a set of stored clusters and $r(c, x)$ and evaluation between

x and cluster c (i.e. distance to center or likelihood of being a member). [1]

To learn f_{ind}, is suffices to learn accurate $r_{ind}(x,c) = \mathbb{1}[r(x,c) > 1]$. Note that in fact many functions r, upon linear scaling, have equivalent level curves, and thus learning optimal r_{ind} is equivalent to optimizing over r over several function classes that have similar level curve properties.

As an example, for fixed function $\ell : \mathcal{D} \times \mathcal{C} \to \mathbb{R}$ and any strictly increasing functions $d_1, d_2 : \mathbb{R} \to \mathbb{R}$, $\exists \omega$ s.t. $\mathbb{1}[\omega \cdot d_1(\ell(x,c)) > 1] = \mathbb{1}[d_2(\ell(x,c)) > 1]$, $\forall x, c$. Allowing ω to be implicitly learned through cluster parameters of c, we have that even optimizing r_{ind} over a simple function class can learn powerful underlying representations of the data.

With this in mind, we choose to associate each cluster to an ellipse and define $r_{ind}(x,c)$ as whether x is in the ellipse corresponding to c. Note that this definition of r_{ind} in fact includes r which models a Gaussian distribution and indeed many other distributions which have elliptic similarity. To ensure monotonocity of f_{ind}, we ensure that ellipses only increase in each axis (radius) and that we never remove ellipses, only adding them. Additionally, to improve ellipse learnability, rather than considering general ellipses, we consider axis-aligned ellipses.

3.2 Cluster Updating

How do we determine how to update the cluster set $\mathcal{C}$. Moreover, as we assume no priors on the existing space, what are effective methods and objectives to initialize and optimize the clusters.

To minimize the given loss, we choose to optimize the "density" of each cluster in the embedding space, aiming to, given a central point, compute the most dense ellipse with this center. Each key is modeled as having certain volume to ensure the optimal ellipse is non-trivial, with the per-key volume dependent on α in the loss function. We empirically find that this objective of optimizing density, in high dimensional space, leads to very conservative clusters, as increasing the radius in multiple directions drastically increases the total ellipse volume. However, we also find that multiple clusters will emerge and grow sufficiently to adapt to patterns with sufficient examples.

As TruAdapt receives new input keys in the embedding space, the model classifies the key based on whether it is "in" an existing ellipse, "close" to an existing ellipse (but not "in"), or "distant" if sufficiently far from any ellipse. As we do not know the true optimal cluster size, each ellipse will start small. For close points to an ellipse, this indicates we may be able to optimize density by increasing the ellipse radii in the direction of the close point. We will add close points to the underlying bloom filter as they are not in any cluster and thus f_{ind} will mis-classify them. For "in" points, as f_{ind} correctly classifies them, we will not edit any ellipse or add them to the bloom filter. For "distant" points, we will center an ellipse at that key embedding with probability $\frac{k}{\max(n, 2^{|\mathcal{C}|})}$, with $\mathcal{C}$ being the existing ellipse (cluster) set.

This choice of sampling ensures we expect approximately $O(\log(n))$ ellipses, which ensures we have insert and test performance of $O(\text{embed\_dim} \cdot \log(n))$. Also, for a new patterns of size $O(n)$, we expect to center an ellipse in that pattern with high probability. Moreover, as empirically validated, with proper classification of "close", we can ensure the ellipse grows only for sufficiently large patterns. These results will be shown in the next section.

For "close" points, we update the ellipse axis radii in each dimension by a multiplicative factor inversely proportional to the relative density of that ellipse and inversely proportional to the points' deviation on that axis. Thus we optimize density by expanding for nearby points while recognizing the existing density relative to the surrounding space. This parameter is especially important as ellipses are initialized with minimal radii and thus, if there is a nearby pattern, the model will notice many more "close" points than "in" points, indicating the current radii are low. Thus, when we have few "in" points yet many "close" points, we quickly increase ellipse radii in a manner similar to exponential search on the radii, allowing us to initiate each ellipse with minimal radii and the ellipse will quickly find adequate radii for each dimension, searching exponentially until the relative density increases. As a proxy for inverse relative density, we maintain a running total of "close" points compared to "close" and "in" points.

3.3 The Complete Algorithm

In the following pseudocode, we will consider x as the embedding of the key. Additionally, note that dilation, rather than being single value, gives maximal performance when represented as a function of the ellipse. Dilation corresponds to how far from an ellipse we search for "close" points, and thus we define dilation with exponential decay based on the ellipse size to ensure larger ellipses search less far. Note that, for the ellipse inverse density parameter (IR_density), as we discuss above, this is simply the ratio of "close" points to "close" and "in" points, which is maintained by the *insert* method, which correlates with the key-point density in the surrounding space compared to in the ellipse.

The programming implementation of this algorithm is in python and assisted by the Scikit-Learn library [7] and the BloomFilter implementation from [2]

3.4 Alternatives and Optimizations

In certain cases, such as for performance independent of n or for increased throughput, we can provide modifications of the

[1]I will note we assume the embedding space has been designed with Euclidean Distance, but with sufficient normalization, the provided algorithms will also work for other distances or similarity metrics, such as dot-product similarity and cosine-similarity

Algorithm 2 TruAdapt

```
Class TruAdapt
Method __init__
    Input: k, min_r, filter_size, hash_funct_num, p
    Initialization:
        $\mathcal{C} = [\,]$  //ellipses
        filter = BloomFilter(filter_size, hash_funct_num)

Method in_ellipse: $r_{ind}$
    Input: x, ellipse
    Return $\sum_i \left(\frac{x_i - ellipse.center_i}{ellipse.radii_i}\right)^2 \leq 1$

Method close_to_ellipse
    Input: x, ellipse
    Return $\sum_i \left(\frac{x_i - ellipse.center_i}{ellipse.dilation \cdot ellipse.radii_i}\right)^2 \leq 1$

Method $f_{ind}$
    Input: x, dilation
    Compute $e^* = \text{argmax}_{e \in ellipses} r_{ind}(x, e, dilation)$
    Return $r_{ind}(x, e^*, dilation), e^*$

Method insert
    Input: x, key
    Evaluate:
        $in\_ellipse, e^* = f_{ind}(x)$
        if $in\_ellipse$: end
        filter.insert(x)
        if not $in\_ellipse$ and $close\_to\_ellipse(x, e^*)$ :
            $e^*$.update($x$)
        else w.p $\frac{k}{\min(n, 2^{|\mathcal{C}|})}$: $\mathcal{C}$.append($ellipse$.init($x$))

Method query
    Input: x, key
    Return: $f_{ind}(x)_0$ or filter.contains(key)
```

Algorithm 3 Ellipse Class Definition

```
Class Ellipse
Attributes:
    center, radii, IR_density

Method __init__
    Input: x
    Initialization:
        center= $x$
        radii= $[0.01, \ldots, 0.01]$ //arbitrary small initiation

Method update
    Input: x
    Evaluate
        $d_i = \left(\frac{x_i - ellipse.center_i}{ellipse.radii_i}\right)^2, \ \forall i$
        $\text{radii}_i = \text{radii}_i \cdot (1 + \gamma \cdot \text{IR\_density} \cdot \text{d}_i), \forall i$
        // $\gamma = 0.61$ determined empirically
```

above algorithm that provide $O(1)$ insert and query or other $O(\log n)$ models with modeling simplifications to improve performance. One natural modification of the algorithm is, rather than considering ellipses, we consider spheres, thus storing fewer parameters per ellipse and simplifying update and compute logic. As most of the computational cost is in the *in_ellipse* method and the ellipse updates, this simplification offers a significant run-time boost while preserving practical modeling capacity.

An alternative algorithm can be used to ensure $O(1)$ memory by ensuring we maintain a limited amount of clusters. We may sample ellipse centers identically as before, but when the amount of clusters reaches an upper bound, we cover the two closest ellipses with a larger ellipse to ensure monotonicity and decrease the ellipse count. However, for certain distributions, even the two closest ellipses can be quite distant, leading to subpar performance. In practice, as typically one data pattern will be represented by several ellipses, it is these ellipses which merge, which maintains accurate representations

An additional optimization we implement to improve throughput is by transforming high dimensional inputs into lower dimensional space. As we have no priors on the distribution, we achieve this via the Johnson-Lindenstrauss transformation on the embeddings [1]. We implement this optimization for improved visualizations and to improve testing speed.

3.5 Performance Guarantees

Theorem 1 ($O(\log(n))$ Clusters)**.** *Given n inserts in the TruAdapt algorithm, with probability* 0.99*, we will have at most* $2k\log(n)$ *clusters, with k being the above sampling parameter.*

For n inserts, listed $x_1, x_2, \ldots, x_n$, consider for any r, the amount of inserts between r and $2r-1$ inclusive. We will aim to bound the maximum amount of new clusters created from the inserts $x_r, \ldots, x_{2r-1}$. First, we suppose that all points are "distant", as "close" and "in" points will not form new ellipses.

Additionally, we will assume there have already been $|\mathcal{C}| \geq \log(2r-1)$ clusters, as otherwise we can simply consider the subset of inserts from x_r to x_{2r-1} that occur after this condition has been met, which will lead us to consider fewer potential inserts.

Thus, as $2^{|\mathcal{C}|} \geq 2r-1$, for each considered insert x_p for $r \leq p < 2r$, the probability of x_p initiating a new cluster is $\frac{k}{p}$. As $p \geq r$, for each x_p, the probability of a new cluster initiated from x_p is bounded above by $\frac{x}{r}$, and thus the expected number of inserts is bounded above by $r \cdot \frac{k}{r} = k$. As inserts occur independently at random, we then have that by the Chernoff bound, with $\mathcal{C}_r$ referring to the additional ellipses from $x_1, \ldots, x_r$

$$\mathbf{Pr}[|\mathcal{C}_{2r-1}| - |\mathcal{C}_{r-1}| \leq 2k] \leq e^{-k/3}$$

Thus, if for any r, $\mathcal{C}_r \geq \log(2r-1)$, we expect the number of inserts from r to $2r-1$ to be less than $2k$ with high probability. As we can decompose $x_1, \ldots, x_n$ into $\log(n/r)$ sequences of the form $x_a, \ldots, x_{2a-1}$, we provide that, with high probability via union bounding on the above probability: $|\mathcal{C}_n| \leq \log(2r-1) + 2\log(n/r)k + C$ for all r and thus $|\mathcal{C}_n| \leq 2k\log(n)$.

Theorem 2 ($O(\log(n))$ Clusters)**.** *For a new pattern of size $\frac{4.7n}{k}$, TruAdapt initiates a cluster at a point in the pattern with 0.99 probability.*

Let $I \subset [n]$ be the index set containing $4.7n/k$ examples of a given pattern. As inserts x_i for $i \in I$ are part of a new pattern, while we have not initiated a new cluster in the region, TruAdapt classifies each point at "distant". For TruAdapt to not initiate a cluster in the pattern, each insert in x_i for $i \in I$ cannot initiate a cluster despite being marked as "distant". Distant points initiate a new cluster with probability at least $\frac{k}{n}$. Thus the failure probability is bounded above by:

$$\left(1 - \frac{k}{n}\right)^{|I|} = \left(1 - \frac{k}{n}\right)^{4.7n/k} \leq e^{-4.7} < 0.01$$

4 Over-Eager Representations

Empirically we have determined an issue with the existing methodology of constructing Learned Bloom Filters. The authors in [2,6] leverage Random Forest models to power their learned filters and consider other model classes, such as neural network models. However, these models, when considering the regions which exceed a threshold, often learn generous space-partition schemes that over-classify the true key region in the input space and render these models vulnerable to distribution shifts. In specific, a query distribution shift into a region incorrectly classified as containing keys can lead to a high FPR.

We show empirically with real data sources that traditional model architectures classify a significantly larger region of the embedding space as a key than is needed, rendering these models vulnerable to distribution shifts that now query non-keys in this misclassified region.

We show in contrast that TruAdapt's cautious model of the key-space leads to conservative modeling that avoids much of the "over-eager" misclassifications of common model families.

We believe the "over-eagerness" of common model classes derives from the limited amount of true false negative training examples compared to the vastness of the input space. Typically in training, the false negatives come from storing negative queries, which is typically from a limited region of the input space and requires significant overhead to acquire.

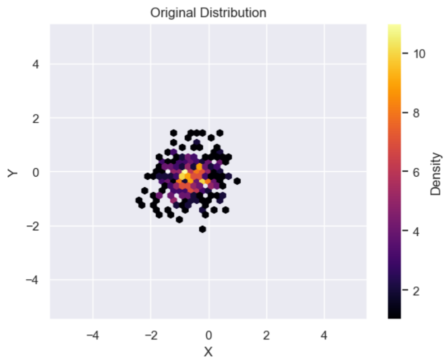

Figure 2: True Data Distribution

TruAdapt does not need negative samples due to its cautious learning of the key distribution.

To demonstrate this feature, we embed the ImageNet class and consider the performance of classifiers TruAdapt, a Random Forest model, Gradient Boosted regression trees, and a fully connected Neural Network with 3 128-parameter hidden layers and ReLU activation. TruAdapt is not given any positive or negative test samples while the other models are trained on 3612 examples, 25% of which are positive. The bias towards negative examples is used to decrease the "eagerness" of these learned models, which we will still show are "over-eager". TruAdapt learns via the 395 insertions. Note also that, for visualization, we JL-project to 2-dimensions.

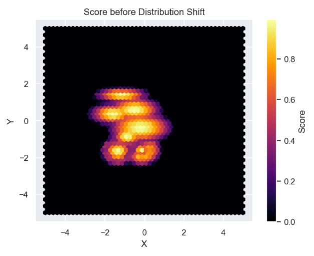

Figure 3: TruAdapt Heatmap

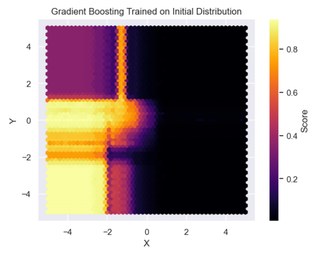

Figure 4: Gradient Boost Heatmap

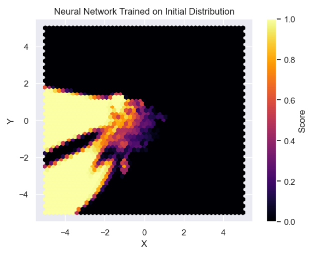

Figure 5: Neural Network Heatmap

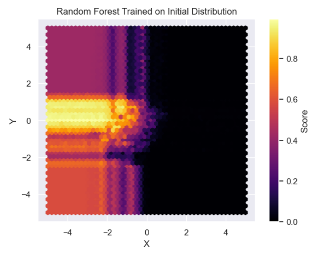

Figure 6: Random Forest Heatmap

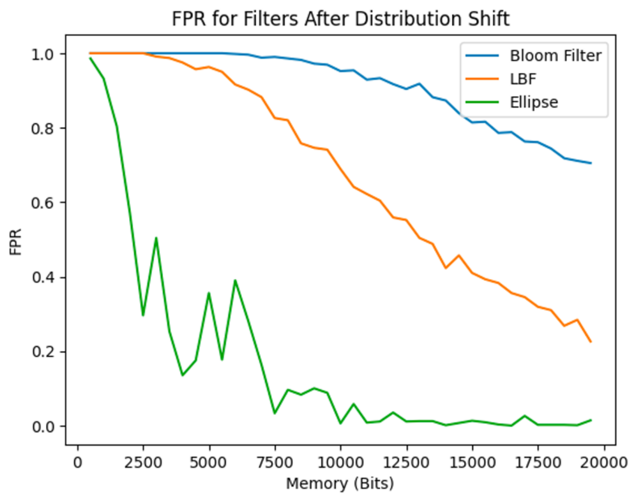

Figure 7: FPR varying with total bloom filter memory for different bloom filter models under a synthetic workload with a distribution shift. Note: Ellipse refers to an earlier name for TruAdapt

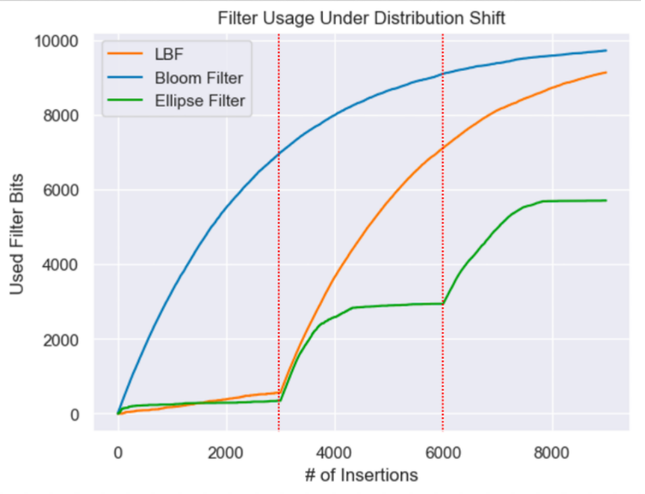

Figure 8: Under a simulated workload with mixture of Gaussian key patterns, modeling the heavy-insertion of a few topics, our filter handles distribtion shifts and even pre-shift matches the performance of LBF. Distribution shifts occur at 3000 and 6000 key insertion, indicated by the dotted red lines. Lower filter usage is better.

Note that in the other three models, despite the true distribution only occupying a small region of the space, the learned models classify sizable regions of the entire key-space as keys. If the distribution of queries shifts to a over-eagerly classified region, the models will have high false positive rate.

5 Evaluations

In our evaluations section, we benchmark our performance against a Learned Bloom Filter (LBF) with a Random Forest classifier and a standard Bloom Filter. We first tested on a synthetic dataset of Gaussians in 10 dimensional space. The initial distribution consisted of points in 10-D space drawn from Gaussians of radii 1 with centers at $\mathbf{0}_{10}$, $\mathbf{10}_{10}$, $\mathbf{20}_{10}$, and $\mathbf{30}_{10}$. The first shifted distribution consisted of points drawn from 10D Gaussians of radii 1 with centers at $-\mathbf{10}_{10}$, $-\mathbf{200}_{10}$, $-\mathbf{30}_{10}$, and $\mathbf{50}_{10}$, and the second from centers at $-\mathbf{40}_{10}$, $-\mathbf{70}_{10}$, $-\mathbf{80}_{10}$, and $\mathbf{224}_{10}$. For all models, we use $k = 4$ hash functions. As we insert the values, we measure space usage for each of the filters. After inserting all of the values, we sample values not in the filter to calculate the false positive rate (FPR) curves against max filter size. Our filter has lower FPR than both the LBF and standard Bloom Filter, as illustrated in Figure 7. Furthermore, we have comparable space usage performance to a LBF in the initial distribution and need only half the space of a LBF after the distribution shift, as seen in Figure 8.

Next, we tested on embeddings obtained from real-world ImageNet data.We obtained 512 dimensional embeddings by removing the last layer of a pretrained ResNet18 model. Due to the difficulty of visualizing data in high-dimensional space, we JL transform our real-world data embeddings down to dimension 2 for our empirical evaluations. However, our method generalizes to embeddings in higher dimensional spaces. We let the dog class be the initial image distribution. As such, we trained a Random Forest classifier images from the dogs class as positive examples and images from the truck, fish, and house classes as negative examples, where each class had 955 examples. For our test set, we chose a 395 images from the original distribution (dogs) and created a distribution shift of truck, church, and gas pump classes, each other 955 examples. After inserting embeddings from the initial distribution into TruAdapt, we plotted score heatmap for TruAdapt to visualize the learned distribution, along with the true distributions of the embeddings (Figure 9). Our method is able to adapt to a distribution shift, unlike LBFs, while also not overeagerly expanding in embedding space.

We then repeat the FPR and filter usage experiments on ImageNet data. When testing the FPR rate, we sampled "not in filter" points within a circle of radius 2 at centered at $(-4, 2)$. We chose to sample these values because they fall outside both the original and shifted distribution but are still relatively close, which penalizes overeager learned distributions. As seen in Figure 10, we have a significantly lower false positive rate than both the Bloom Filter and the LBF. In fact, the LBF often has worse performance than the Bloom Filter due to false positives from an overeager learned distribution. Moreover, we use 4.5x less space than an LBF and 7x less space than a Bloom filter for a fixed insertion workload with a

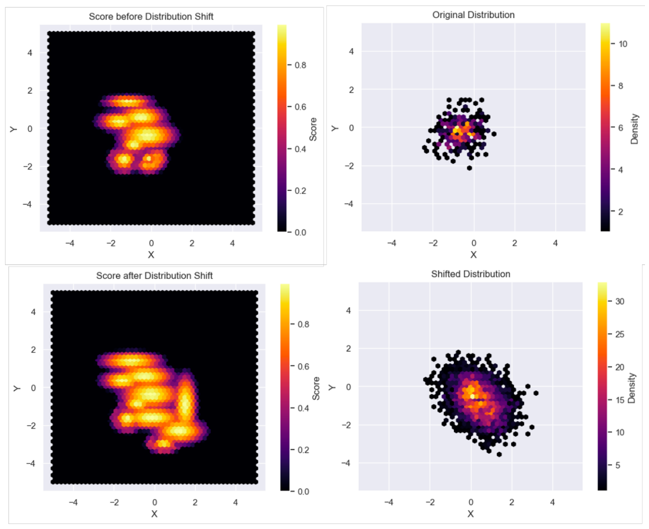

Figure 9: The heatmaps on the left represent the learned distribution of our model, while the heatmaps on the right are the actual frequency distribution of inserted keys. Our model accurately expands and adapts to distribution shifts in inserted keys.

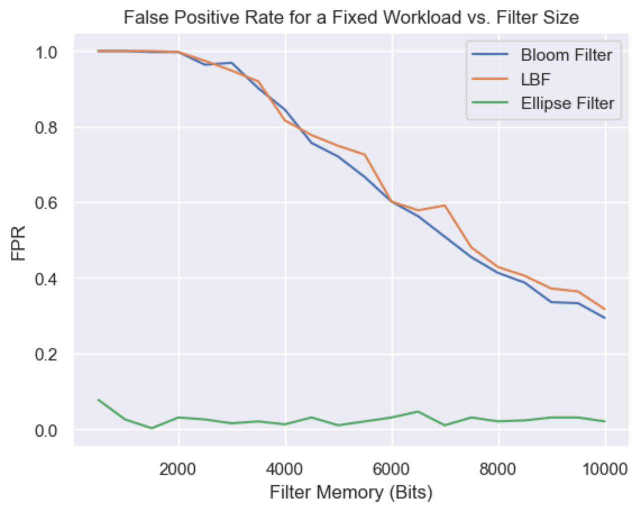

Figure 10: FPR varying with total bloom filter memory for different bloom filter models under a real-world image embedding workload with a distribution shift.

distribution shift (Figure 11). As such, we have demonstrated that TruAdapt has improved empirical performance over both Bloom Filters and LBFs on both synthetic and real-world key-insertion workloads. Without a distribution shift, we have comparable performance to a standard LBF, and have vastly improved performance when there is a distribution shift, due to our non-eager learning process.

6 Conclusion

TruAdapt succeeds in producing adaptive models suitable to the necessary guarantees of Bloom Filters and can adapt other model classes in producing accurate models of a changing key distribution. The core advantages of TruAdapt lie in its ability to adaptively model the key space with no priors: able to model an unknown distribution and applicable to cases where training sets do not exist, as is common in bloom filter workloads. Additionally, in its cautious modeling, TruAdapt avoids the pitfalls of many common model classes, which produce an over-eager space representation, rendering these classification models vulnerable to heavy key misclassification in certain distribution shifts. Additionally, we provide performance guarantees and propose optimizations and additional algorithms to handle varying performance specifications. Lastly, we test TruAdapt with real data drawn from ImageNet, validating the intuitive design nature with impressive empirical performance.

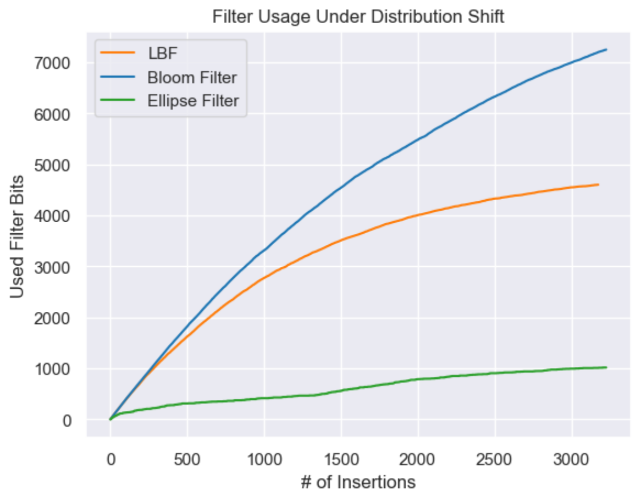

Figure 11: Under a real-data workload of image embeddings, TruAdapt (ellipse filter) out performs both LBF and a standard Bloom Filter. Distribution shifts occur at approximately 350 insertions, but is less evident in the chart when compared to 8 because the distribution shift is not as drastic.

7 Collaboration Statement

Devan Shah worked on algorithm design, studying failures such as "over-eagerness" in the LBF systems, and producing the underlying algorithm for TruAdapt in addition to several failed algorithms (e.g. the Semantic Hash filter). Devan implemented the TruAdapt algorithm and designed the optimization criteria in conjunction with literature from streaming algorithms and recommendations systems. Devan worked on model understanding from an intuitive and theoretical standpoint and led the paper writing and poster writing.

David Yan empirically validated the model on synthetic and real datasets, studying representations and patterns in common data sets for insights on model development and accurate pattern modeling, produced figures to showcase model performance and illustrate LBF shortfalls, led experiment design and fine-tuned model parameters, such as γ and the dilation function, through rigorous ablations and extensive testing.

Of course, as we both worked in the same room together during the duration of the project, many ideas and contributions are shared.

References

[1] Chen, L. Johnson-lindenstrauss transformation and random projection, 2023.

[2] Dai, Z., and Shrivastava, A. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier. *CoRR abs/1910.09131* (2019).

[3] Google Developers. Collaborative filtering, 2023.

[4] Grathwohl, W., Wang, K.-C., Jacobsen, J.-H., Duvenaud, D., Norouzi, M., and Swersky, K. Your classifier is secretly an energy based model and you should treat it like one, 2020.

[5] Kerbl, B., Kopanas, G., Leimkühler, T., and Drettakis, G. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics 42*, 4 (July 2023).

[6] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. *CoRR abs/1712.01208* (2017).

[7] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.